A Extension to Other Tasks

In this paper, we focused on the task of rough terrain navigation of wheeled robots. As such, some of our implementation details are based on task specific assumptions. However, we believe the proposed approach is also applicable to other tasks. In this section, we describe how task specific assumptions could be removed to extend the algorithm to any POMDP problem.

One limitation of our algorithm is the availability of trajectory tracking controllers. In our experiments, we used the well known PD trajectory tracking controller for Ackermann steered robots described in Sec. B. However, this controller may not be applicable other types of systems. We note though that trajectory trackers can be formed for any system using LQR (Linear Quadratic Regulator) as long as there exists a differentiable state transition model [35, 36, 37]. The neural network state transition models learned in our approach can be differentiated numerically or using backpropagation.

Another limitation is that we assume full state knowledge. Although the dynamics models in our implementation made state transition predictions using observations, we assumed the robot knew its initial full state and used a task specific observation model to calculate observations given predicted states. This assumption restricts the algorithm to MDP problems. However, recent methods [38, 39] have formed predictive dynamics models for partially observable systems using latent-space models. These approaches learn encoders to map observations to latent vectors. These latent vectors encode the underlying state of the system. A latent dynamics model is also learned to predict state transitions in the latent space. Using such approaches could extend our proposed method to POMDP problems.

B PD Trajectory Tracker

In this section, we define the function f_{track} that corresponds to the PD trajectory tracking controller used in our experiments. In Sec. 5, we defined trajectory tracker as a controller that provided corrective actions given a reference trajectory $\bar{d} = (\bar{s}_0, ..., \bar{s}_T, \bar{a}_0, ..., \bar{a}_{T-1})$, and the robot's actual states $s_0, ..., s_t$, where $t < T$. So the actual action applied to the robot is

$$a_t = \bar{a}_t + f_{track}(s_0, ..., s_t, \bar{d}), \tag{8}$$

. The PD tracker operates by calculating the error from the robot's true state s_t and the corresponding reference state from the reference trajectory $\bar{s}_t$. The calculated error and the change in its value is then used to calculate a corrective action.

In our implementation, we define the error vector $e_t \in \mathbb{R}^2$ as the vector from the actual state s_t to the reference state $\bar{s}_t$ relative to the robot's frame and projected to the ground plane, as shown in Fig 5. The corrective action can then be calculated with

$$f_{track}(s_0, ..., s_t, \bar{d}) = K_p e_t + K_d(e_t - e_{t-1}), \tag{9}$$

where $K_p \in \mathbb{R}^{2\times2}$ and $K_d \in \mathbb{R}^{2\times2}$ are positive diagonal gain matrices. In this formulation, an action of $a = [1, 0]^T$ will drive the robot forward in the $+x$ direction, and an action of $a = [0, 1]^T$ will steer the robot counterclockwise in the $+\theta_z$ direction as depicted in Fig 5.

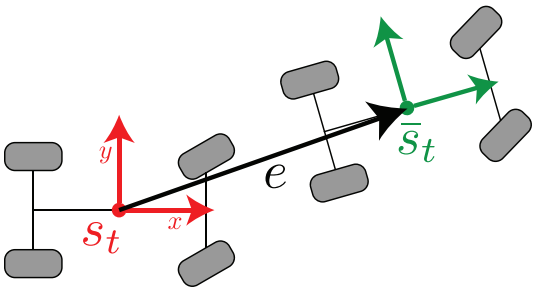


Figure 5: Definition of PD tracker's error vector. The actual state s_t is shown in red. The reference state $\bar{s}_t$ is shown in green. The error vector is shown in black and defined relative to frame s_t.

C Predicted Distribution & Negative Log-Likelihood Estimates

In our implementation, the dynamics model predicts the state transition, denoted as $\dot{s}_t \in \mathbb{R}^{\dot{n}}$, given the robot's observation, denoted as o_t, and action, denoted as a_t. We define the state transition function, which maps the robot's current state $s_t \in \mathbb{R}^n$ and state transition $\dot{s}$ to the robot's next state s_{t+1}, as

$$s_{t+1} = \mathcal{F}(s_t, \dot{s}_t). \tag{10}$$

We also define its inverse, which maps the robot's current and next state to the state transition, as

$$\dot{s}_t = \mathcal{F}^{-1}(s_t, s_{t+1}). \tag{11}$$

Robot observations are directly generated from the state using the observation function

$$o_t = O(s_t). \tag{12}$$

The prediction from the dynamics model takes the form of an uncorrelated Gaussian distribution. The output of the neural network's last layer is the predicted distribution's mean $\mu \in \mathbb{R}^{\dot{n}}$ and log variance log-$\sigma \in \mathbb{R}^{\dot{n}}$. We denote the dynamics model's mapping of the observation and the action to the predicted mean and log variance as

$$\mu_t, \text{log-}\sigma_t = \hat{p}_\theta(o_t, a_t) \tag{13}$$

Given the robot's predicted current state $\hat{s}_t$ and action a_t, the negative log-likelihood of the robot's true next state s_{t+1} can be found as follows: 1) A robot observation $\hat{o}_t$ is generated using $\hat{s}_t$ and equation 12. 2) The state transition distribution's mean μ_t and log variance log-σ_t is predicted using $\hat{o}_t$, a_t and equation 13. 3) The target state transition $\dot{s}_t$ is found using $\hat{s}_t$, s_{t+1} and equation 11. 4) The negative log-likelihood is found with

$$\mathcal{L} = \frac{1}{2}\log(2\pi^{\dot{n}}) + \frac{1}{2}\sum_{i=1}^{\dot{n}}\left[\frac{([\mu_t]_i - [\dot{s}_t]_i)^2}{\exp([\text{log-}\sigma_t]_i)} + [\text{log-}\sigma_t]_i\right], \tag{14}$$

where $[.]_i$ represents the i^{th} element of a vector.

When using a multistep loss $\mathcal{L}_{ms}^N$ with $N > 1$, the negative log-likelihood of next state s_{t+1} is found by averaging the likelihood of s_{t+1} over N particles $\hat{s}_t^1, ..., \hat{s}_t^N$, then taking the negative log. For numerical stability, we avoid calculating this directly. Instead, we calculate the negative log-likelihood of s_{t+1} given each particle using equation 14, denoted as $\mathcal{L}_1, ..., \mathcal{L}_N$. Then, with $\mathcal{L}_{min} = \min_i \mathcal{L}_i$,

$$\mathcal{L}_{ms}^N = \mathcal{L}_{min} - \log\left(\frac{1}{N}\sum_{i=1}^{N}\exp(\mathcal{L}_{min} - \mathcal{L}_i)\right) \tag{15}$$

D Neural Network Architecture & Training Considerations

For the simulated environment, the input to the dynamics model included a $\mathbb{R}^{1x64,64}$ terrain map, and a $\mathbb{R}^9$ robot state vector. The output was a $\mathbb{R}^{26}$ state transition distribution mean and log-variance vector. Table 5 shows the neural network architecture used for simulation experiments. For the real world environment, the input to the dynamics model included a $\mathbb{R}^{1x16,16}$ terrain map, and a $\mathbb{R}^3$ previous state transition vector. The output was a $\mathbb{R}^6$ state transition distribution mean and log-variance vector. Table 6 shows the neural network architecture used for real world experiments.

Using a multistep loss for dynamics model training is important for proper uncertainty propagation. However, training with a multistep loss can be computationally intensive and may lead to instability when single step predictions are inaccurate. In our implementation, we initialized training using a single step loss. We then change to a multistep loss using multistep trajectories that were progressively increased in length. The longest trajectory used during multistep training had a length of 5 steps. These trajectories were cropped from much longer trajectories. Earlier portions of the longer uncropped trajectories were used to initialize the hidden state of the LSTM layer.

Table 5: The network architecture for simulation

Layer	Input	Input Dim	Activation	Output	Output Dim
$CNN_{1,8}$	Heightmap	$1 \times 64 \times 64$	ReLU	Hmap feat1	$8 \times 32 \times 32$
$CNN_{8,4}$	Hmap feat1	$8 \times 32 \times 32$	ReLU	Hmap feat2	$4 \times 16 \times 16$
$Reshape$	Hmap feat2	$4 \times 16 \times 16$	None	Hmap feat3	1024
$FC_{1037,2048}$	Hmap feat3 + robot state	1037	ReLU	feat1	2048
$LSTM_{2048,256}$	feat1	2048		feat2	256
$FC_{256,256}$	feat2	256	ReLU	feat3	256
$FC_{256,13}$	feat3	256	None	pred. μ, log-σ	13

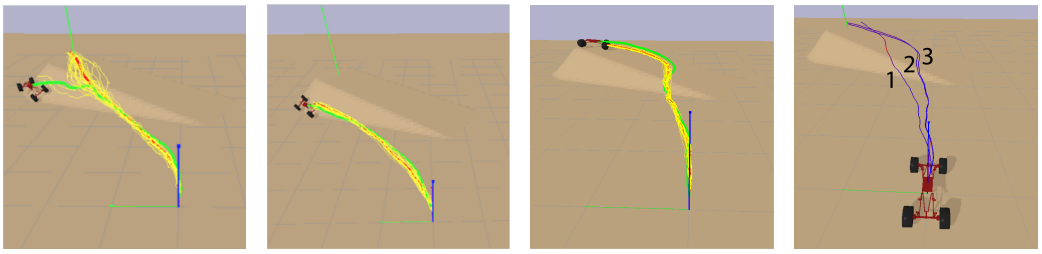
Table 6: The network architecture for real world

Layer	Input	Input Dim	Activation	Output	Output Dim
$CNN_{1,8}$	Heightmap	$1 \times 16 \times 16$	ReLU	Hmap feat1	$8 \times 8 \times 8$
$Reshape$	Hmap feat1	$8 \times 8 \times 8$	None	Hmap feat2	512
$FC_{515,2048}$	Hmap feat2 + prev. transition	515	ReLU	feat1	2048
$LSTM_{2048,256}$	feat1	2048		feat2	256
$FC_{256,256}$	feat2	256	ReLU	feat3	256
$FC_{256,13}$	feat3	256	None	pred. μ, log-σ	3

E Additional Simulation Results

Figure 6 shows a trial in an illustrative simple ramp environment. When divergence is not considered, the optimizer find a short, but highly divergent trajectory up the steep sides of the ramp. The robot fails to execute the trajectory and slides off the ramp due to insufficient friction. When penalizing divergence, the optimizer finds an overly conservative trajectory that does not attempt to go up the ramp. Our divergence constrained method allows the robot to consistently climb up the ramp and reach the goal.

More trials in randomly simulated environments are shown in Figure 7.



(a) No uncertainty (b) Penalty (c) Constraint (d) AL Optimization

Figure 6: Ramp climbing trial. The optimized, predicted closed-loop, and actual trajectories are shown in red, yellow, and green respectively. The optimization process for (c) is shown in (d), where the divergence constraint is progressively satisfied, with red indicating higher divergence.

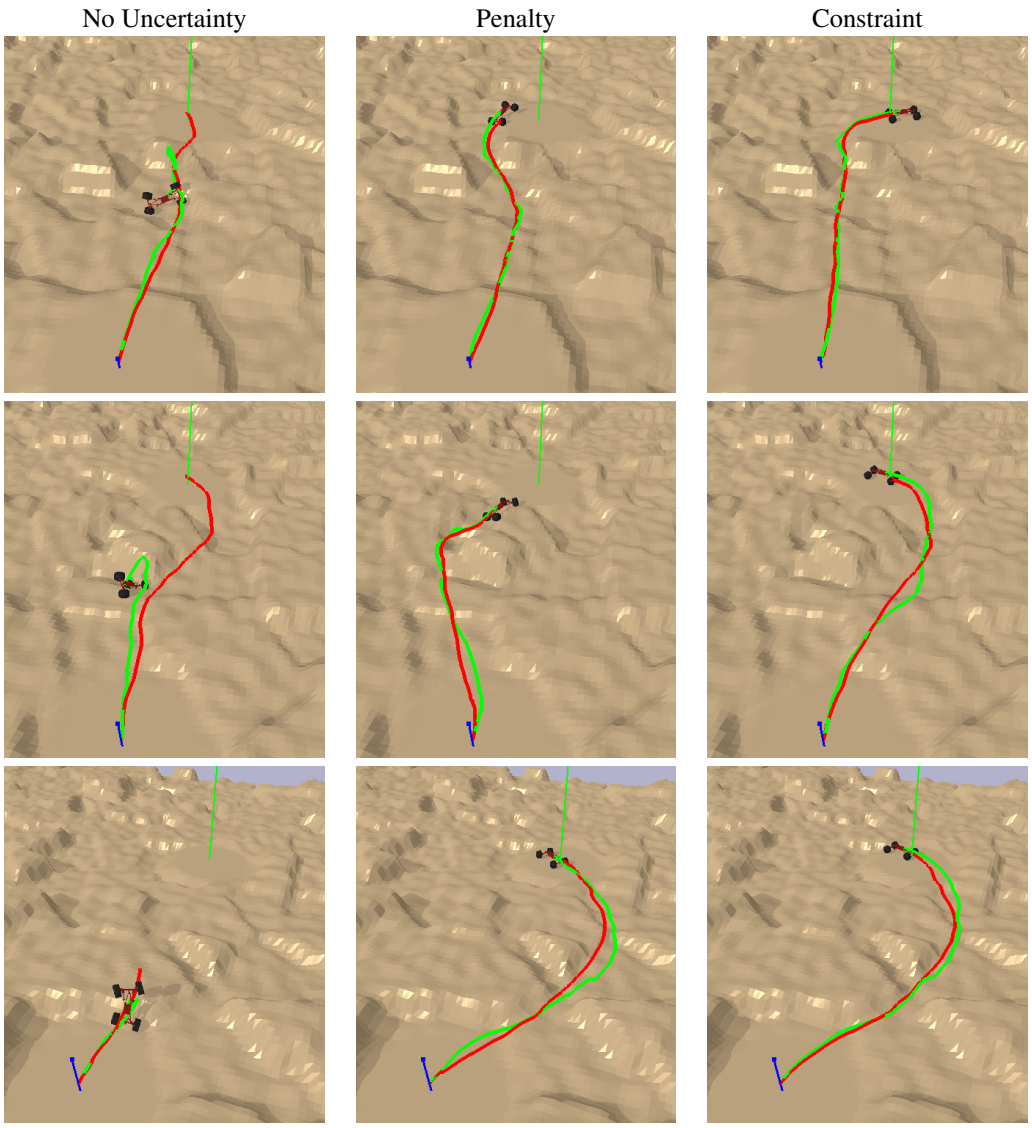


Figure 7: Additional results on random terrain